Incremental View Maintenance (IVM)

06/20/2024 - Matt Wonlaw, Rocicorp

Why talk about IVM @ a Local-First event?

Apps are views over data

```ruby
# Rails
User.joins(:tags)
    .where(tags: { name: ['Brunette', 'Impolite'] } )
    .group('users.id')
    .having('count(*) = 2')
```

```php
// Laravel
$users = DB::table('users')
            ->join('contacts', 'users.id', '=', 'contacts.user_id')
            ->join('orders', 'users.id', '=', 'orders.user_id')
            ->select('users.*', 'contacts.phone', 'orders.price')
            ->get();
```

But times have changed

- In the Rails and LAMP days, sites were mostly request-response
  - Full page refresh and full query re-run on modification
- Today: apps have long persistent sessions with little to no page reloads.
  - Query results need to be updated as things change.
  - Queries are subscriptions

Example

```
function IssueList({workspace}) {
  const issues = useQuery(
    `SELECT issue.*, user.name FROM issue
      WHERE workspace_id = ?:workspace
      JOIN user ON issue.owner_id = user.id
      ORDER BY modified DESC LIMIT 100`,
      {workspace}
  );

  return (
    <table>
      <TableHeader />
      {issues.map(issue => <IssueRow issue={issue} />)}
    </table>
  );
}
```

As the state in the database changes, the `IssueList` should automatically render the update without us doing anything. No refresh. No re-running the query.

Introducing IVM

- Think of it as:
    - A data dependency graph (like signals) plus
    - Incremental computation over collections
- E.g., `map` / `reduce` / `filter` but without making copies and by only running compute over modified items

Without IVM

```
x.map(..).filter(..).reduce(..);
```

Any modification to `x` results in `N` array copies.

Without IVM

```
const issues = Array.from({length: 100_000}, genIssues);
const fooIssues = issues.filter((issue) => /.*foo.*/.test(issue.title));
```

Any modification of `issues` re-scans `100_000` items

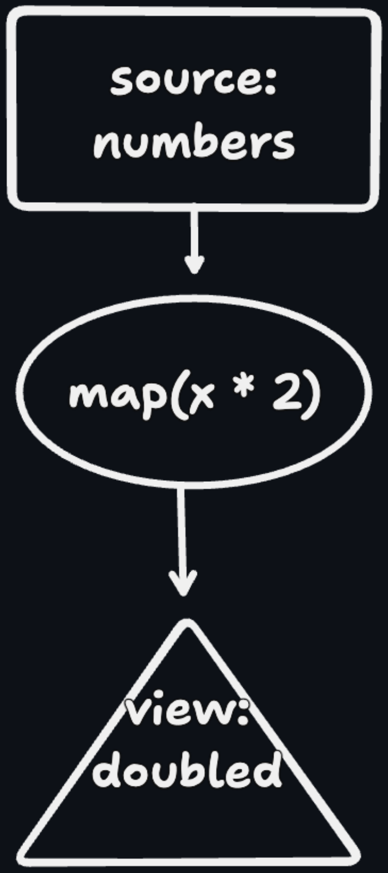
Implementing IVM

Terminology

- **Pipeline** - a chain of computations.
  E.g., `x.map.filter.reduce`
- **Source** - the data provider and root of the pipeline.
  E.g., the `x` in `x.map.filter.reduce`
- **Operator** - an internal node in a pipeline.
  E.g., `map` / `filter` / `reduce` / `join`
- **View** - the final result of a pipeline.
  E.g., `const view = x.map.filter.reduce`
- **Difference** - a change sent through the pipeline

Simple Pipeline

```
doubled = numbers.map(x => x*2)
```



source: numbers

map(x * 2)

view: doubled

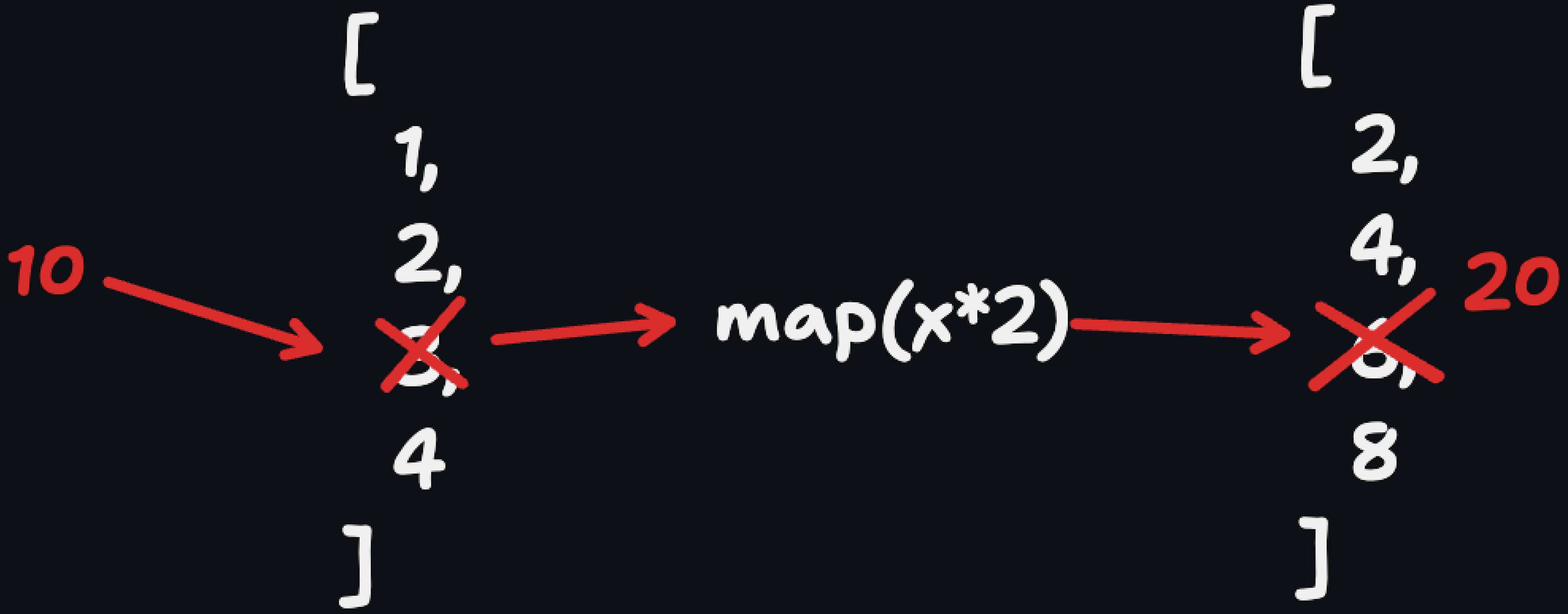
Incremental `map`

```
numbers.map(x => x*2)
```

- Map already is incremental
- Only depends on a single, not the entire collection
- The problem is preserving a relationship between `source` and `view`

Incremental `map`

```
numbers.map(x => x*2)
```



Preserving View & Source Relationship

- A modification of an input in the source should modify the corresponding output
- Options:
    i. Make the `source` and `view` each a `Map<K, V>` to associate items with a key

    ii. Make the `source` and `view` take a `comparator<S, V>`
- Option (1): Diff events take the form of `[key, value]`
- Option (2): Diff events are still just the `value`

Tracking Deletes, Updates, Adds

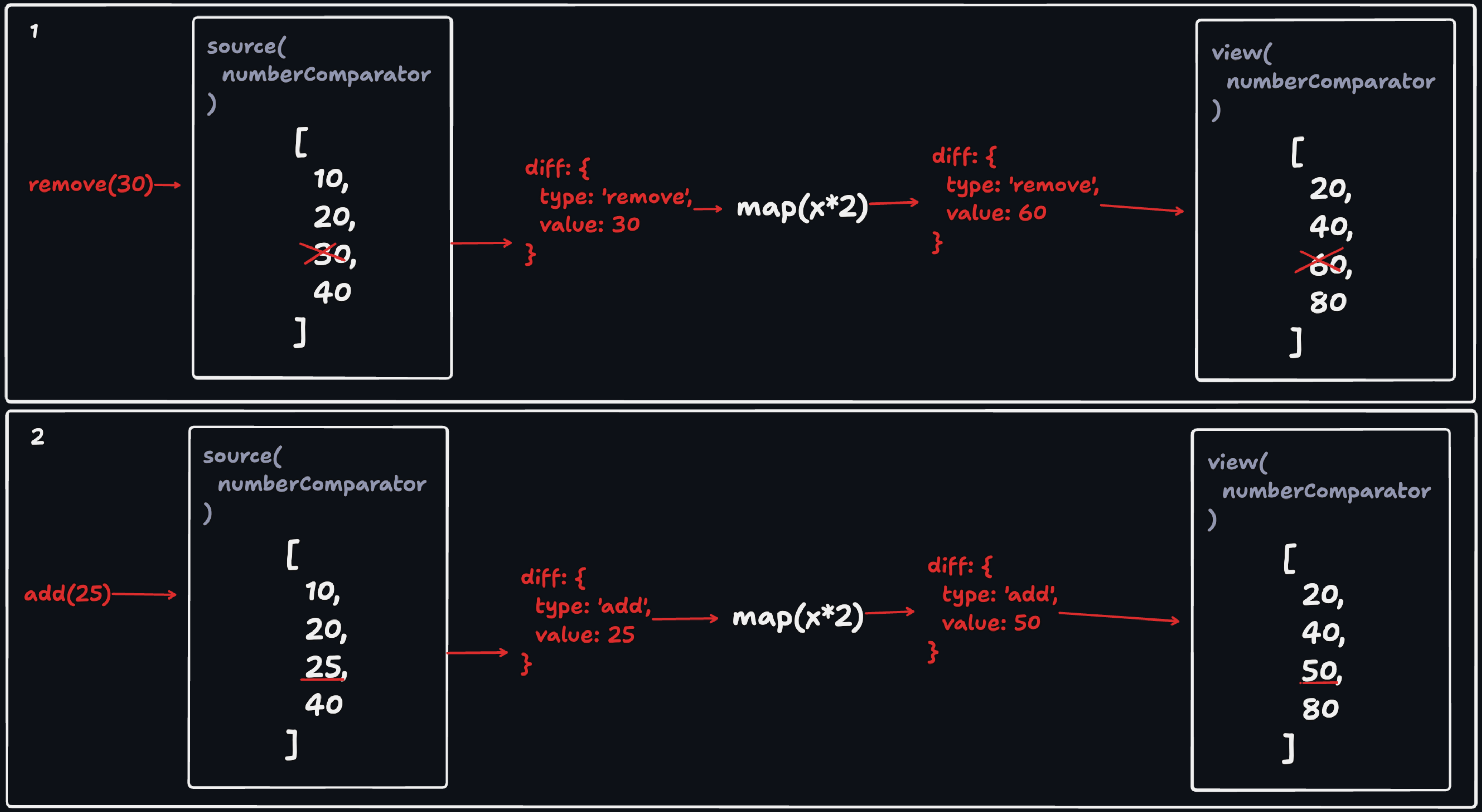
- Add an event `type` to the difference event

```
type DifferenceEvent<T> = {
  type: 'add' | 'remove',
  value: T
};
```

- `update` is modeled as `remove` followed by `add`

Putting it together

replace 30 with 25



1

remove(30)→

source(
  numberComparator
)

[
  10,
  20,
  ~~30~~,
  40
]

→ diff: {
  type: 'remove',
  value: 30
}
→ map(x*2) →

diff: {
  type: 'remove',
  value: 60
}
→

view(
  numberComparator
)

[
  20,
  40,
  ~~60~~,
  80
]

2

add(25)→

source(
  numberComparator
)

[
  10,
  20,
  25,
  40
]

→ diff: {
  type: 'add',
  value: 25
}
→ map(x*2) →

diff: {
  type: 'add',
  value: 50
}
→

view(
  numberComparator
)

[
  20,
  40,
  50,
  80
]

Notes:

1. `filter` can be implemented similarly
2. `join` and `reduce` operators need to consider past values. Require "memory" to make them incremental.
3. Duplicate entries in a collection can be handled by adding a multiplicity to difference events.

```
type DifferenceEvent<T> = {
  multiplicity: number; // -N: remove N times. +N: add N times. 0: no-op.
  value: T;
};
```

How can we create an incremental query language with incremental `map/reduce/filter/join`?

Example: Modeling SQL

SELECT = `map`

```
// SELECT title FROM issue
issues.map(i => i.title)
```

WHERE = `filter`

```
// SELECT * FROM issue WHERE priority = 1
issues.filter(i => i.priority = 1)
```

WHERE .. AND .. = `filter.filter`

```
// SELECT * FROM issue WHERE priority = 1 AND status = 1
issues.filter(i => i.priority = 1).filter(i => i.status = 1)
```

Example: Modeling SQL

WHERE .. OR .. = `x.filter.concat(x.filter).distinct`

```
// SELECT * FROM issue WHERE priority = 1 OR status = 1
issues.filter(i => i.priority = 1).concat(issues.filter(i => i.status = 1)).distinct(i => i.id)
```

GROUP BY (or any aggregation) = `reduce`

```
// SELECT * FROM issue GROUP BY status
issues.reduce((acc, issue) => {
  const existing = acc.get(issue.status);
  if (existing) {
    existing.push(issue);
  } else {
    acc.set(issues.status, [issue]);
  }
}, new Map())
```

Example: Modeling SQL

JOIN = `join`

Custom operator that correlates two streams. Conceptually:

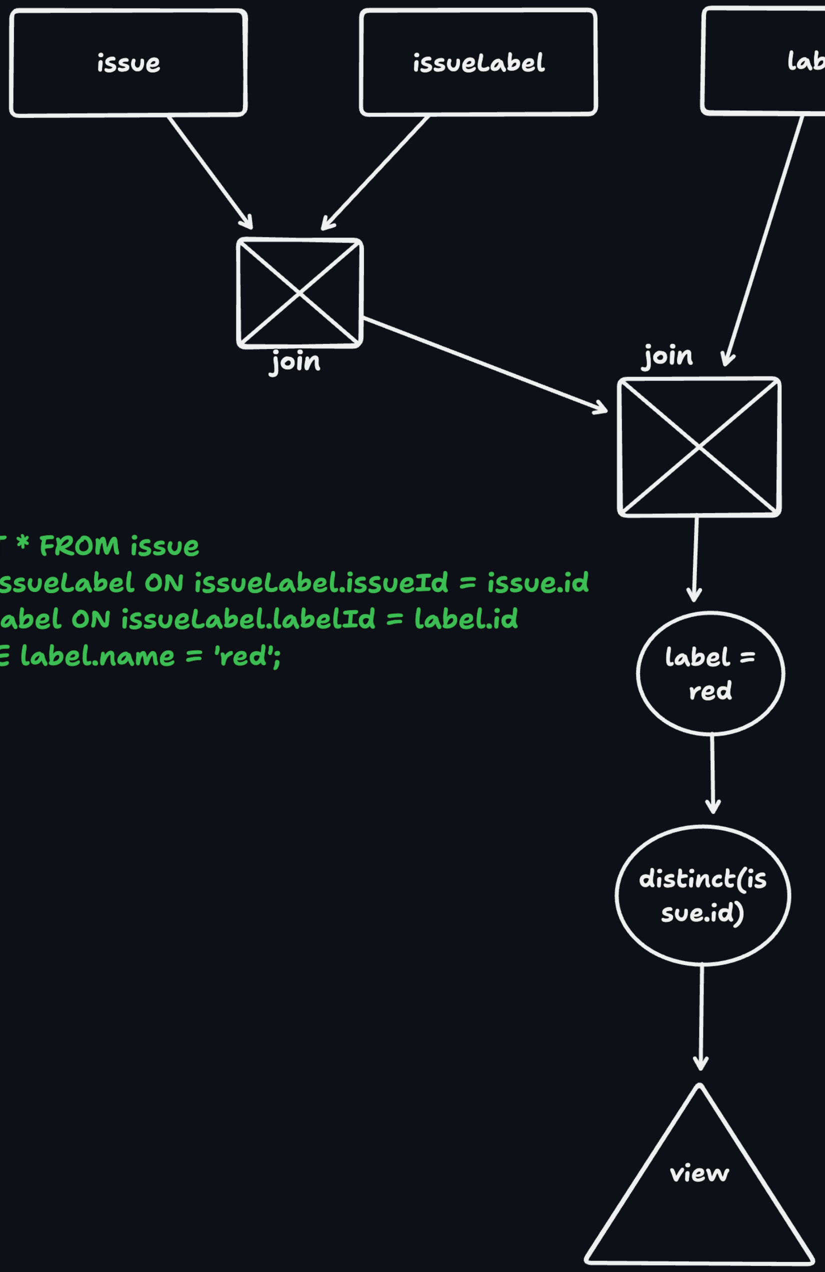
```
// SELECT * FROM issue JOIN user ON issue.owner_id = user.id
issues.map(issue => {
  const user = users.get(issue.owner_id);
  if (user) {
    return {
      issue,
      user
    };
  }
  return undefined;
}).filter(row => row !== undefined)
```

But updated to handle `difference events`, be incremental and respond to changes in either the `user` or `issue` table.

Modeling SQL

1. Map each language construct to an incremental `filter` / `map` / `reduce` / `join` / `concat` / `distinct` operator
2. Wire these operators together in a DAG

Example DAGs

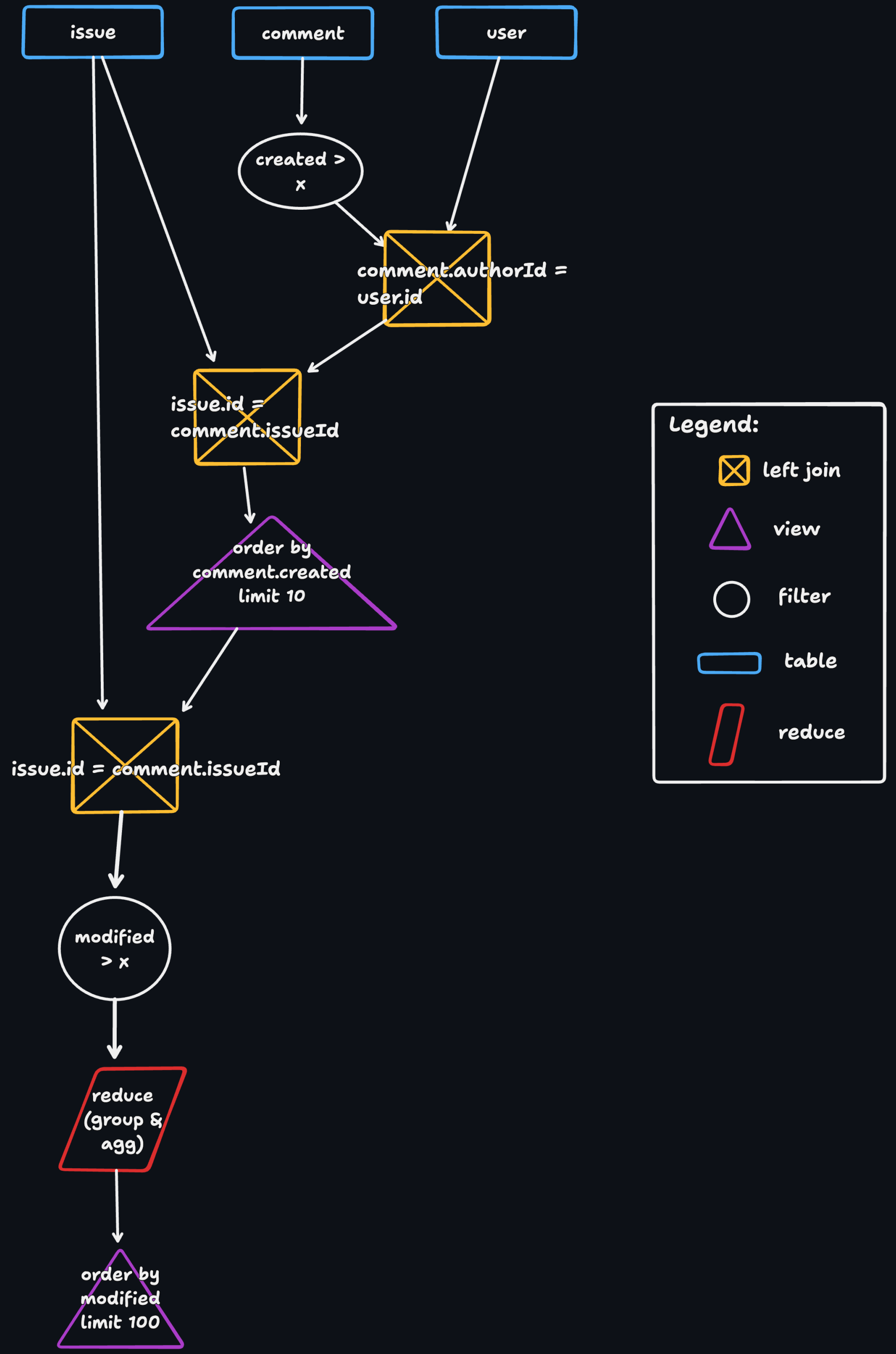


```
SELECT * FROM issue
JOIN issueLabel ON issueLabel.issueId = issue.id
JOIN label ON issueLabel.labelId = label.id
WHERE label.name = 'red';
```

```
SELECT
  title,
  (SELECT
    json_group_array(json_object(
      'body', body,
      'author', (SELECT
        json_group_array(name)
        FROM user WHERE user.id = comment.authorId
      )
    )) FROM comment
    WHERE created > x AND comment.issueId = issue.id
    ORDER BY created ASC LIMIT 10
   ) AS comments
FROM issue WHERE modified > x ORDER BY modified DESC LIMIT 100
```

I.e.,

```
issue.select(
  'title',
  iq => iq.related('comments')
    .where('created', '>', date)
    .select(
      'body',
      cq => cq.related('author').select('name')
    )
    .order('created').limit(10),
).order('modified', 'desc').limit(100);
```


issue
comment
user
created > x
comment.authorId = user.id
issue.id = comment.issueId
order by comment.created limit 10
issue.id = comment.issueId
modified > x
reduce (group & agg)
order by modified limit 100
Legend:
left join
view
filter
table
reduce

Demo

- Linearite 1 mil
- ZQL vs SQLite

Further Topics

- Subqueries
- Recursive queries
- Normalizing queries
- Sharing structure between pipelines for efficiency
- First run & query planning
- Reducing memory consumption of `join` & `reduce`
- Re-ordering the DAG
- Index creation
- Order By & Limit: properties of the `view`

Resources

- Incremental Query Language Theory: https://github.com/vmware-archive/database-stream-processor/blob/main/doc/vldb23/main.pdf
- Signals: http://adapton.org/
- In our new product as "ZQL (Zero Query Language)" https://zerosync.dev/